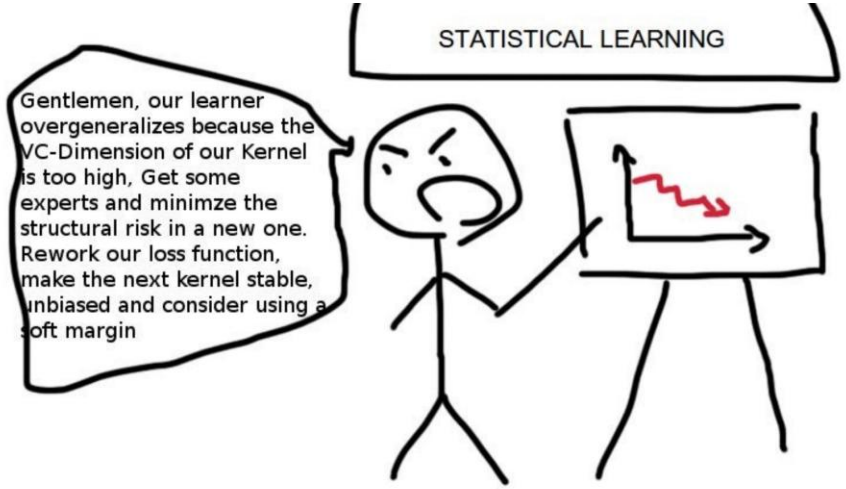
Neural Networks

Gautam Kamath

Onto Neural Networks



Recall: The XOR Problem

- (Draw it on domain $\{0,1\}^2$)
- There is no linear separator which separates the +'s from the –'s
- Can prove it, but I won't
- How can we solve this problem?
- Kernels
  - Apply mapping to data, use linear model on top
  - Can be some generic kernel, "hand-crafted" features (domain expertise), etc.
- Neural Network
  - *Learn* a mapping of data (from data), use linear model on top
  - Learn a *representation*

Drawing some old models

- (Draw generic linear model in graphical form)
- $x \in \mathbf{R}^2, \tilde{y} = x_1 w_1 + x_2 w_2 + b = \langle x, w \rangle + b$
- (Add on sigmoid to output to make into logistic regression)
- $\text{sigmoid}(t) = \sigma(t) = \frac{1}{1+e^{-t}}$
- $\hat{y} = \frac{1}{1+\exp(-\langle w,x \rangle - b)}$ (logistic regression)

Drawing a simple multilayer perceptron

- (Draw 2LNN with width-2 hidden layer, x input, u and c first layer weights and biases, z hidden layer pre-activation, f non-linearity, h hidden layer post-activation, w and b second layer weights and biases, $\tilde{y}$ output)
- (Label input layer, hidden layer, representation layer, output layer, non-linear activation $\mathbf{R} \to \mathbf{R}$)

Some calculations for XOR

- $z = Ux + c, h = f(z), \tilde{y} = \langle h, w \rangle + b$
- Consider: $U = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, c = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, w = \begin{bmatrix} 2 \\ -4 \end{bmatrix}, b = -1$
  - Parameters to be learned, but suppose they're just given for now
- Choose $f(t) = \max(0, t) = \mathrm{ReLU}(t)$ (draw)
  - *Activation function* – this is a hyperparameter choice
- $x_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, y_1 = -1.\ z = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, h = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \tilde{y} = -1$
- $x_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, y_1 = 1.\ z = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, h = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \tilde{y} = 2 - 1 = 1$, etc.
- Adding the non-linear f really gave it a lot more power!

Drawing a bigger multilayer perceptron

- (Draw wider three-layer MLP, input $x \in \mathbf{R}^d$, output $\tilde{y} \in \mathbf{R}^m$)
  - (Illustrate depth and width, representation layer)
- $z^{(1)} = W^{(1)}x, h^{(1)} = f\left(z^{(1)}\right), z^{(2)} = W^{(2)}h^{(1)}, \ldots$
- What to do with output $\tilde{y} \in \mathbf{R}^m$?
  - Put through *softmax* to get distribution over m classes ("confidences" of each)
  - $\hat{y}_i = \frac{\exp(\tilde{y}_i)}{\sum_{j=1}^{m} \exp(\tilde{y}_j)}$
- What loss function? Use the *cross-entropy* loss
  - $\ell_\theta(x, y) = -\sum_{i=1}^{m} y_i \log \hat{y}_i$
    - Use "one-hot encoding" of y: if $y = c$, then $y_c = 1$, and $y_i = 0$ for other entries
    - $\hat{y} = g_\theta(x)$, where g_θ is a (somewhat complicated) function

Activation Functions (Draw)

- Non-linear
- Sigmoid: $\sigma(t) = \frac{1}{1+e^{-t}} = \frac{e^t}{1+e^t}$
- Tanh: $\tanh(t) = \frac{e^t - e^{-t}}{e^t + e^{-t}}$
- ReLU: $\text{relu}(t) = \max(0, t)$
  - Still has a strong gradient signal even if t is large

Training

- Loss function: $\arg\min_{\theta} L = \frac{1}{n}\sum_i \ell_\theta\left(x^{(i)}, y^{(i)}\right)$
  - Recall $\ell_\theta(x, y) = -\sum_{j=1}^{m} y_j \log \hat{y}_j$
- Just use gradient descent!
  - $\theta^t = \theta^{t-1} - \eta_t \nabla L_{\theta^{t-1}}$
- But… how to compute ∇L?
  - Recall $\hat{y} = g_\theta(x)$ is some complicated function… how to take derivative?
  - Luckily computers are very good at this
  - Automatic differentiation, backpropagation algorithm
    - Chain rule + dynamic programming

Backpropagation (simple scalar network)

$$z = wx + b$$
$$h = \sigma(z)$$
$$s = vh + c$$
$$L = \frac{1}{2}(s - y)^2$$

- Forward pass (draw)
- What are the parameters of this model?
  - Calculate derivatives
  - Backwards pass

Backpropagation (multiple neurons)

$$x \in \mathbf{R}^d, W \in \mathbf{R}^{m \times d}, b \in \mathbf{R}^m, v \in \mathbf{R}^m, c \in \mathbf{R}$$
$$z = Wx + b$$
$$h = \sigma(z)$$
$$s = v^T h + c$$
$$L = \ell(s, y)$$

- Do calculations

General backprop (for MLPs)

For $\ell = 1, \ldots, K$,

$$a^{(0)} = x$$

$$z^{(\ell)} = W^{(\ell)} a^{(\ell-1)} + b^{(\ell)}, a^{(\ell)} = \sigma_\ell(z^{(\ell)})$$

$$\delta^{(\ell)} = \nabla_{z^{(\ell)}} L = \left(W^{(\ell+1)}\right)^T \delta^{(\ell+1)} \odot \sigma'_\ell\left(z^{(\ell)}\right)$$

$$\nabla_{W^{(\ell)}} L = \delta^{(\ell)}\left(a^{(\ell-1)}\right)^T, \nabla_{b^{(\ell)}} L = \delta^{(\ell)}$$

Universality of MLPs

- Any continuous function $g: [0,1]^d \rightarrow \mathbf{R}$ can be approximated arbitrarily well by some 2 layer NN with arbitrary non-polynomial activation
- (maybe draw)
- Caveat: may require the hidden layer to be incredibly wide

Overfitting and Generalization

- p parameters, n datapoints, d dimensions
- Classical setting: $p \approx d$
- Modern neural networks: $p \gg nd$
  - Biggest neural networks today: $> 1{,}000{,}000{,}000{,}000$ parameters (1 trillion)
  - MNIST dataset: 60000 images, 28×28 pixels $\rightarrow nd \approx 47$ million
  - Note that it's possible for a network with $\approx nd$ parameters to "memorize" training dataset – no generalization guarantee

Avoiding Overfitting

- Bagging
- Regularization
  - Regularized loss: $L_{\theta_{t-1}}(x, y) + \frac{\lambda}{2}\|\theta_{t-1}\|_2^2$
  - Taking the gradient wrt θ: $\nabla L_{\theta_{t-1}}(x, y) + \lambda\theta_{t-1}$
  - Gradient descent: $\theta_t \leftarrow \theta_{t-1} - \eta(\nabla L_{\theta_{t-1}}(x, y) + \lambda\theta_{t-1})$
  - Equivalently: $\theta_t \leftarrow (1 - \eta\lambda)\theta_{t-1} - \eta\nabla L_{\theta_{t-1}}(x, y)$
  - Sometimes called *weight decay* in neural networks
- Data augmentation

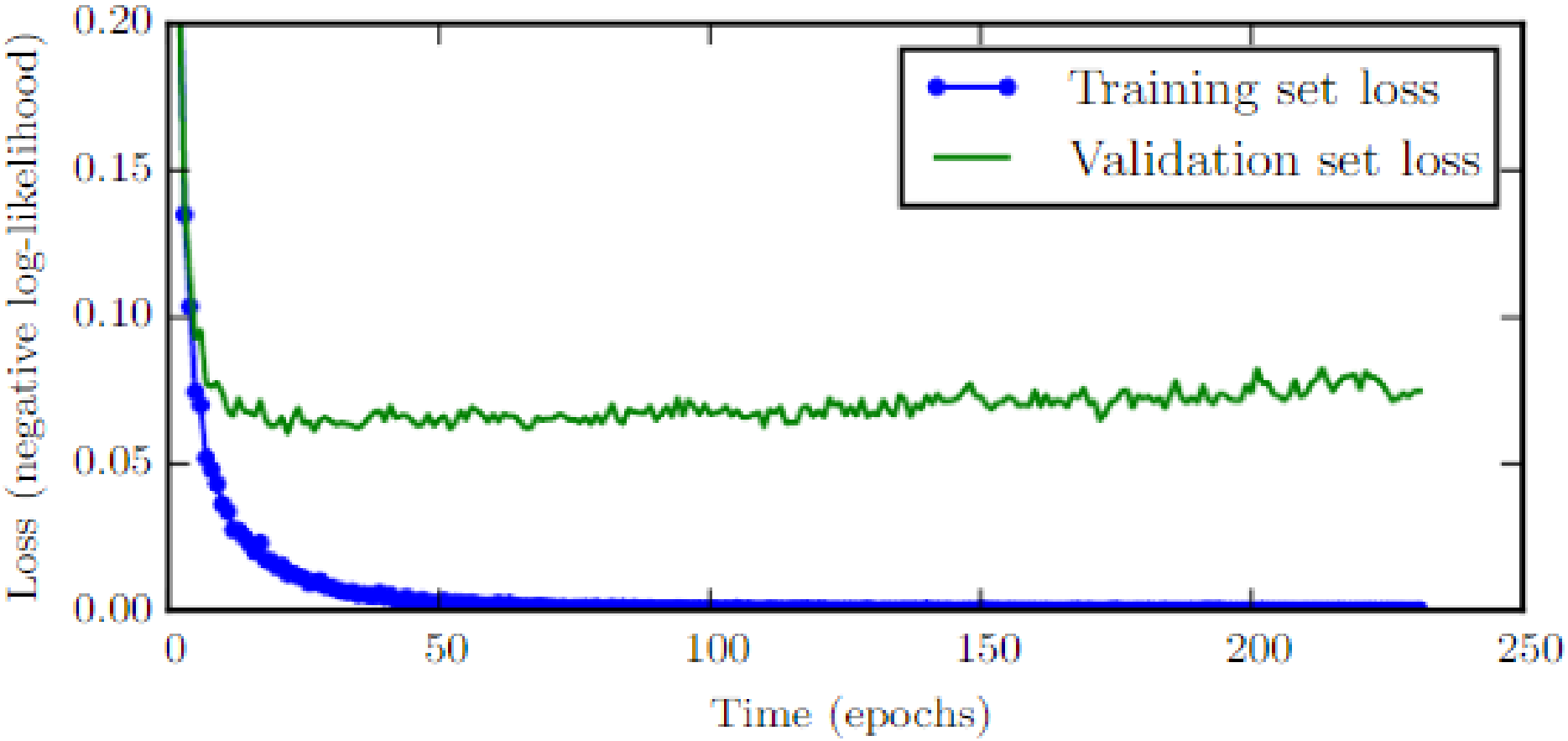
Data Augmentation






- But be careful! (6 becomes 9 when rotated)

Early Stopping



Dropout

- Keep each node w.p. $p > 0$ when training (independent for each point)
- (Draw network, draw dropped out version)
- Common hyperparameters: 0.5 for hidden nodes, 0.8 for inputs
- Consider input to layer 2: $z^{(2)} = W^{(2)}h^{(1)} + b^{(2)}$
  - Since only p fraction of $h^{(1)}$'s are kept (in expectation), must scale up by $1/p$
  - (Draw $h^{(1)}$ layer, with mask and scaling)
- Test time: no dropout, no scaling
  - This is called inverted dropout (more common)
  - Otherwise, scale at test time instead

Normalization

- Normalize features before training
- Compute mean: $\mu = \frac{1}{n}\sum X_i$
- Recenter data around mean: $X_i \leftarrow X_i - \mu$
- Compute variance of each coordinate: $\sigma_j^2 = \frac{1}{n}\sum X_{ij}^2$
- Rescale data in each coordinate: $X_{ij} \leftarrow X_{ij}/\sigma_j$
- (Draw transformation)

Batch Normalization

- $z^{(i)} = W^{(i)} h^{(i-1)} + b^{(i)}$
- $h^{(i)} = f\left(z^{(i)}\right)$
- (draw)
- Normalize the coordinates of $z^{(i)}$ over each minibatch
  - Debate: normalize $z^{(i)}$ or $h^{(i)}$?
- Add scale and shift learnable parameters
- Apply to each neuron individually

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Layer Normalization

- Batchnorm: Average within each neuron, over a batch
- Layernorm: Average within each layer, over single datapoints
- (Draw)

Optimization

- First-order methods: things which use only first derivative info
- Second-order methods: things which use second derivative info
  - Take more memory, time per step, but fewer steps. Less popular in practice.

Standard First-Order Methods

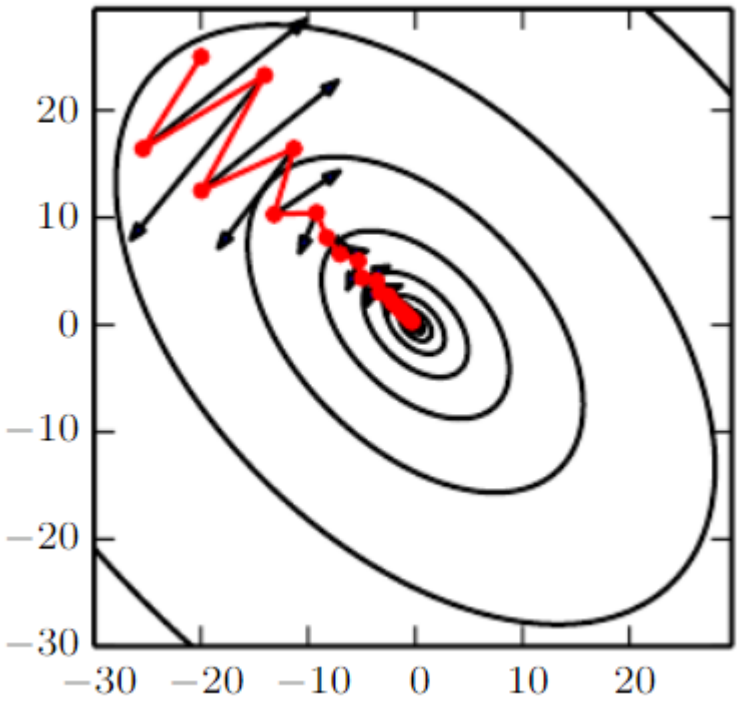
- Batch gradient descent
  - $\theta \leftarrow \theta - \eta \cdot \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \ell_\theta(x_i, y_i)$
  - Attempts to estimate $E_{(x,y)\sim D}[\nabla_\theta \ell_\theta(x_i, y_i)]$
- Stochastic gradient descent
  - $\theta \leftarrow \theta - \eta \cdot \nabla_\theta \ell_\theta(x_i, y_i)$
  - Pick a random example, take a step. Or instead of random: shuffle dataset, go over them in order. Reshuffle after each epoch.
  - Epoch: Going over the entire dataset once
- Minibatch stochastic gradient descent
  - $\theta \leftarrow \theta - \eta \cdot \frac{1}{|B|} \sum_{i \in B} \nabla_\theta \ell_\theta(x_i, y_i)$
  - Minibatch size can be 64, 128, 256, etc.

Challenges

- How to choose η?
- Learning rate schedules don't adapt to data
- Different learning rates for different coordinates?

Momentum

- Keep memory of previous gradient step
- Let $\gamma < 1$ (say = 0.9)
- $v_t = \gamma v_{t-1} + (1-\gamma)\eta \cdot \frac{1}{|B|}\sum_{i \in B} \nabla_{\theta_{t-1}} \ell_{\theta_{t-1}}(x_i, y_i)$
  - New step: weighted sum of old step and current gradient
- $\theta_t \leftarrow \theta_{t-1} - v_t$
- $v_t = 0.1 g_t + 0.1 \cdot 0.9 g_{t-1} + 0.1 \cdot 0.9^2 g_{t-2} + \cdots$
  - Total coefficient $1 - \gamma^t$
- Variant: Nesterov momentum



Adaptive Learning Rates

- Change LR for each parameter over course of optimization, based on how "important" each parameter seems
  - If a coordinate has lots of updates or big updates, lower LR for parameter
  - If a coordinate has few updates or small updates, bigger LR for parameter
- Let $g_t \in \mathbf{R}^p$ be the (estimate of) gradient at time t
- SGD: $\theta_{t,i} \leftarrow \theta_{t-1,i} - \eta g_{t,i}$
- Additionally, define $G_{t,i} = \sum_{j=1}^{t} g_{j,i}^2$ (sum of squared gradients)
- AdaGrad: $\theta_{t,i} \leftarrow \theta_{t-1,i} - \frac{\eta}{\sqrt{G_{t,i}+\varepsilon}} g_{t,i}$
  - ε is a small number
  - Problem: learning rate is penalized "forever," could become tiny

RMSProp

- Previously $G_{t,i} = \sum_{j=1}^{t} g_{j,i}^2$
- Instead, use "momentum" on $G_{t,i}$
- $G_{t,i} = 0.9G_{t-1,i} + 0.1g_{t,i}^2$
  - Replace sum of squared gradients with a weighted sum
  - Will "forget" old gradients over time
- RMSProp: $\theta_{t,i} \leftarrow \theta_{t-1,i} - \frac{\eta}{\sqrt{G_{t,i}+\varepsilon}} g_{t,i}$

Adam

- Use momentum and RMSProp at the same time
  - Plus a bias correction
- $\beta_1, \beta_2, \varepsilon$ hyperparameters
  - $\beta_1 = 0.9, \beta_2 = 0.999, \varepsilon = 10^{-8}$
- $m_{t,i} = \beta_1 m_{t-1,i} + (1 - \beta_1) g_{t,i}$ (momentum)
- $v_{t,i} = \beta_2 v_{t-1,i} + (1 - \beta_2) g_{t,i}^2$ (RMSProp)
- $\widehat{m}_{t,i} = \frac{m_{t,i}}{1-\beta_1^t}, \hat{v}_{t,i} = \frac{v_{t,i}}{1-\beta_2^t}$
- $\theta_{t,i} \leftarrow \theta_{t-1,i} - \frac{\eta}{\sqrt{\hat{v}_{t,i}+\varepsilon}} \widehat{m}_{t,i}$